Fast Track to Scala

Josh Suereth

Typesafe

2012-03-05

Agenda

Why Scala?

Setting up the development environment

First steps

Basic OO features

Testing in Scala

Collections and functional programming

For-expressions and for-loops

Inheritance and traits

Pattern matching

XML support

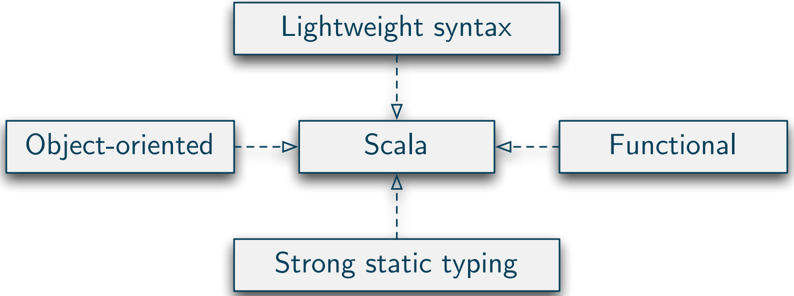
Agenda

- Why Scala?
- Setting up the development environment
- First steps
- Basic OO features
- Testing in Scala
- Collections and functional programming
- For-expressions and for-loops
- Inheritance and traits
- Pattern matching
- XML support

Scala is mature

- 1996 - 1997: Pizza
- 1998 - 2000: GJ, Java generics, javac
- 2001: Scala design begins
- 2003: First experimental release
- 2005: Scala 2.0 written in Scala
- 2006: Industrial adoption starts
- 2007: First release of the Lift web framework
- 2008: First Scala LiftOff unconference, Twitter adopts Scala
- 2009: Big increase in adoption, IDEs mature
- 2010: Scala 2.8 released, first ScalaDays conference
- 2011: Scala 2.9 released, Typesafe Inc. founded

Scala is a unifier



Scala is concise

```scala
class Time(val hours: Int, val minutes: Int)
```

```java
public class Time {                                    // Java
  private final int hours;
  private final int minutes;
  public Time(int hours, int minutes) {
    this.hours = hours;
    this.minutes = minutes;
  }
  public int getHours() {
    return hours;
  }
  public int getMinutes() {
    return minutes;
  }
}
```

Scala is expressive

```
scala> val numbers = 1 to 10
... = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> numbers filter { _ > 5 }
... = Vector(6, 7, 8, 9, 10)
```

Scala is fully interoperable with Java

```
scala> import org.slf4j.LoggerFactory
import org.slf4j.LoggerFactory

scala> val logger = LoggerFactory.getLogger("logger")
logger: org.slf4j.Logger = Logger[logger]

scala> logger.info("Hello!")
09:25:11.393 [Thread-7] INFO logger - Hello!
```

Agenda

Why Scala?

Setting up the development environment

First steps

Basic OO features

Testing in Scala

Collections and functional programming

For-expressions and for-loops

Inheritance and traits

Pattern matching

XML support

Scala distribution

Exercise: Install the Scala distribution

- Download the latest stable release 2.9.1 as an archive for your platform from www.scala-lang.org/downloads
- Unpack the archive to a suitable location, e.g. *~/tools/scala/*
- Add the *bin/* directory to your path
- Verify the installation by opening a terminal and entering *scala -version*:

```
tmp$ scala -version
Scala code runner version 2.9.1.final ...
```

- Also download the Scala API documentation, unpack and browse it

Exercise: "Hello World!" on the command line

- Create the file *Hello.scala*[1] using an arbitrary text editor:

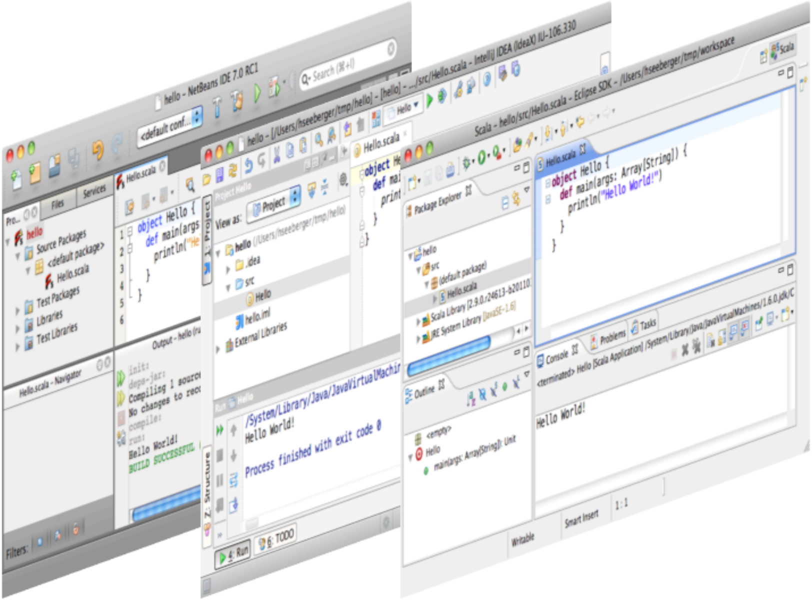
```scala
object Hello {
  def main(args: Array[String]) {
    println("Hello World!")
  }
}
```

- Compile and run it:

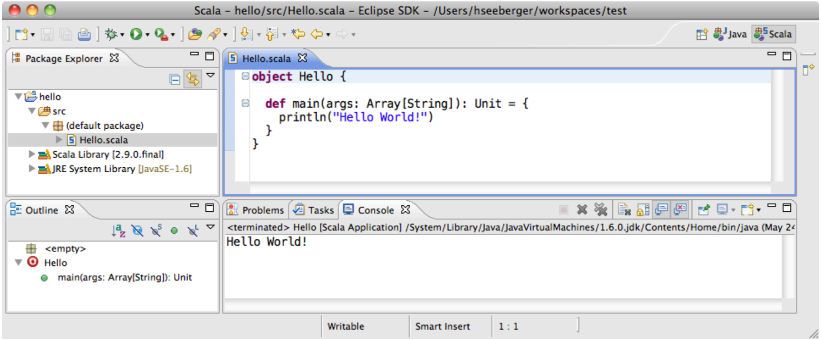
```
tmp$ scalac Hello.scala
tmp$ scala Hello
Hello World!
```

[1] You don't have to understand the code yet!

There are plugins for all major IDEs



Scala IDE for Eclipse



- We, the trainer(s), will use Eclipse for this course
- Feel free to use another IDE or none at all, but we will only be able to offer limited support

Exercise: Install Eclipse and the Scala IDE for Eclipse

- Download and install Eclipse Helios (3.6) or Indigo (3.7) Classic (!) for your platform from www.eclipse.org/downloads
- Install the Scala plugin via the menu *"Help > Install New Software ..."* using the update site http://download.scala-ide.org/releases-29/stable/site
- Verify the installation by opening a fresh workspace, e.g. *~/workspaces/training-scala/*, and switching to the Scala perspective

Exercise: "Hello World!" in Eclipse

- Create a *"New > Scala Project"* with name *hello*
- Create a *"New > Scala Object"* with name *Hello*
- Copy the code from the previous exercise
- Select *"Run As > Scala Application"* from the context menu of the editor or package explorer
- In order to avoid conflicts with other future projects we suggest you now close or delete this project

Simple Build Tool (sbt)

```
tmp$ cd scalatrain
scalatrain$ sbt
[info] Set current project to default (in build file:/Users/hseeberger/.sbt/plugins/)
[info] Set current project to default (in build file:/Users/hseeberger/tmp/scalatrain/)
> compile
[success] Total time: 0 s, completed May 24, 2011 1:14:42 PM
```

- THE build tool for Scala
- Writen in Scala and specifically for Scala
- Used by most real-world projects

Exercise: Install sbt

- Download the launcher:
  http://repo.typesafe.com/typesafe/ivy-releases/org.scala-tools.sbt/sbt-launch/0.11.2/sbt-launch.jar
- Create the following file as a start script for sbt:
  - *sbt* on Mac/Linux:

    ```
    java -Xmx512M -jar <LAUNCHER-JAR> "$@"
    ```

  - *sbt.bat* on Windows:

    ```
    java -Xmx512M -jar <LAUNCHER-JAR> %*
    ```

Exercise: Create a sbt project

- Create a fresh project directory, e.g. *~/projects/training-scala/*, and cd into it
- Attention: Do not create this in your Eclipse workspace!
- Starting sbt will take you to an interactive session
- Execute the following three commands at the sbt prompt:

```
> set name := "scalatrain"
...
> set scalaVersion := "2.9.1"
...
> session save
...
```

- Take a look at the new file *build.sbt* in the project directory *training-scala/*
- Keep the sbt session running!

sbt commands - quick overview

- General commands:
  - *exit* ends the current session
  - *help* lists available commands
- Build commands:
  - *compile* compiles main sources
  - *test:compile* compiles test sources
  - *test* runs tests
  - *console* starts the REPL
  - *run* looks for a main class and runs it
  - Triggered execution: Prefix a command with $\sim$
- Other commands:
  - *clean* deletes all output in the *target/* directory
  - *reload* reloads the build

Exercise: Install the sbt-Eclipse integration

- The sbteclipse plugin let's you create Eclipse project files from an sbt project
- In the project directory (*training-scala/*), create the subdirectory *project/* and there the file *plugins.sbt* with the following contents:

```
addSbtPlugin("com.typesafe.sbteclipse" %
    "sbteclipse-plugin" % "2.0.0")
```

- Attention:
  - Copy and paste is your friend, but sometimes the quotes are not copied correctly!
  - Also, use only one line!

Exercise: Create Eclipse project files

- In the sbt session execute the commands *reload* and the now available *eclipse*

```
> reload
...
> eclipse
...
[info] Successfully created Eclipse project files ...
```

- Import the new Eclipse project using *"Import..."* > *"Existing Projects into Workspace"*
- Verify the import by inspecting the project, e.g. the source directories *src/main/scala/* etc.

Agenda

Interactive programming in the REPL

- In a terminal window enter *scala* to start the REPL
- Alternatively, in a sbt session enter the command *console*

```
tmp$ scala
Welcome to Scala version 2.9.1.final ...
Type in expressions to have them evaluated.
Type :help for more information.
```

- The REPL will evaluate Scala code:

```
scala> "Hello " + "World!"
res0: java.lang.String = Hello World!
```

- Enter *:quit* or *:q* to exit the REPL:

```
scala> :q
```

Immutable variables

- Scala encourages us to use immutable objects:
  - Code free of side-effects is easier to reason about
  - Pure functions can be tested easily
  - Immutable objects won't lead to concurrency issues
- An immutable variable is defined with *val*:

```
scala> val message = "Hello " + "World!"
message: java.lang.String = Hello World!
```

- An immutable variable is ... immutable:

```
scala> message = "Won't compile!"
<console>:8: error: reassignment to val
      message = "Won't compile!"
```

Type inference

- Why does this code compile?

```scala
val message = "Hello " + "World!"
```

- Because in most cases the compiler is able to infer the types
- Of course we can be explicit and use a type annotation:

```scala
val message: String = "Hello " + "World!"
```

- And of course Scala is statically typed:

```scala
scala> val message: Int = "Hello " + "World!"
<console>:7: error: type mismatch;
 found   : java.lang.String("Hello World!")
 required: Int
```

Mutable variables

- Sometimes we really need mutable state
  - This holds true less often than OO programmers might believe
  - As a proof watch out how often we will use mutable state ...
- A mutable variable is defined with *var*:

```
scala> var year = 2011
year: Int = 2011
```

- A mutable variable can be ... mutated:

```
scala> year = 2012
year: Int = 2012
```

Everything has a value

- The last expression of a code block determines its value:

```
scala> val block = {
     |   val x = 1
     |   val y = 2
     |   x + y
     | }
block: Int = 3
```

- *if-else* has a value[2]:

```
scala> :type if (1 == 2) "weird" else "correct"
java.lang.String
```

[2] *:type* shows the type of an expression without evaluating it

Methods

- A method is defined with *def*:

```
scala> def add(x: Int, y: Int) = x + y
add: (x: Int,y: Int)Int
```

- The type should be given for public or non-trivial methods:

```
scala> def add(x: Int, y: Int): Int = x + y
add: (x: Int,y: Int)Int
```

Procedures aka *Unit*-Methods

- The type *Unit* means that the method's value doesn't matter:

```
scala> def sayHello(): Unit = println("Hello!")
sayHello: ()Unit

scala> sayHello
Hello!
```

- There is a special syntax for procedures[3]:

```
scala> def sayHello() {
     |   println("Hello!")
     | }
sayHello: ()Unit
```

[3] Actually Martin Odersky thinks, that this special case for procedures was a mistake; therefore we recommend you better don't use it.

Uniform access principle

- A method without parameters can be written without parens:

```
scala> def message = "Hello World!"
message: java.lang.String
```

- Convention: No-parens style only for side-effect-free methods
- Then there is no distinction for the client between a field (stored value) and a method (computed value)

Operators are methods

- In Scala everything is an object!
- Operators are methods with one parameter used in (dot-less) operator notation:

```
scala> "a,b,c" split ","
res0: Array[java.lang.String] = Array(a, b, c)
```

- Almost all characters are allowed for (method) identifiers:

```
scala> def *?!(s: String) = s.reverse
$times$qmark$bang: (s: String)String
```

- Therefore the following is the call of the method $+$ on the object *1* with the argument *2*:

```
1 + 2
```

A first glance at functions

Just to see where we will be going to:

```
scala> val numbers = List(1, 2, 3)
numbers: List[Int] = List(1, 2, 3)

scala> numbers map (x => x + 1)
res0: List[Int] = List(2, 3, 4)

scala> numbers sortWith ((x, y) => x > y)
res1: List[Int] = List(3, 2, 1)

scala> numbers map (_ + 1) sortWith (_ > _)
res2: List[Int] = List(4, 3, 2)
```

Agenda

Classes

- Classes are blueprints for objects
- Use the keyword *class* to define a class:

```scala
class Train
```

- This is valid Scala code:
  - No semicolon thanks to semicolon inference
  - No access modifier, because public visibility is default
  - No curly braces, since *Train* has no body yet
- Classes have public visibility by default

Creating class instances

- Use the keyword *new* and the name of a class to create an instance:

```scala
new Train
```

- Actually you are calling the primary constructor which results from the class definition
- This is valid Scala code: No parens needed for an arity-0[4] constructor

[4] Arity-n means n arguments.

Exercise: Create the class *Train*

- Use the directory *src/main/scala/*
- Use the file name *Train.scala*
- When done, create an instance in the REPL:

```
scala> val train = new Train
train: Train = Train@77aa89eb
```

Class parameters

- Use parens after the class name to define class parameters:
  - A single parameter is defined by its name, followed by a colon and its type
  - Multiple parameters are separated by comma

```scala
class Train(number: String)
```

- Class parameters result in parameters of the primary constructor[5] and are not visible from the outside

[5] They are only retained as private fields if used in methods or lazy vals.

Exercise: Add a class parameter to *Train*

- Parameter name: *number*
- Parameter type: *String*
- Try to create a *Train* like before:

```
scala> :replay
Replaying: new Train
<console>:8: error: not enough arguments for
    constructor Train: (number: String)Train.
Unspecified value parameter number.
```

- Try to create a *Train* giving a *number*:

```
scala> val train = new Train("Nighttrain")
train: Train = Train@72d53ac7
```

Auxiliary constructors

- Use the keywords *def* and *this* to define auxiliary constructors:

```scala
class Train(number: String) {
  def this() = this("Default")
  def this(n1: String, n2: String) = this(n1 + n2)
}
```

- An auxiliary constructor must immediately call another constructor of the class using *this*
- Tip: In many cases named and default arguments[6] are preferrable

[6] Coming soon, please have a little patience ;-)

Immutable fields

- Use the keyword *val* to define an immutable field:

```scala
class Train(number: String) {
  val kind = "ICE"
}
```

- Fields have public visibility by default:

```scala
scala> val train = new Train("Nighttrain")
train: Train = Train@76d88b7b

scala> train.kind
res0: java.lang.String = ICE
```

Mutable fields

- Use the keyword *var* to define a mutable field:

```scala
class Train(number: String) {
  var kind = "ICE"
}
```

- Of course you can change the value of a mutable field:

```scala
scala> val train = new Train
train: Train = Train@64dff6e3

scala> train.kind = "CHANGED"

scala> train.kind
res0: java.lang.String = CHANGED
```

Make fields out of class parameters

- Also use *val* (or *var*) to make a field out of a class parameter:

```scala
class Train(val kind: String, val number: String)
```

- Now you can access the "converted" class parameters:

```scala
scala> val train = new Train("ICE", "722")
train: Train = Train@3b1674f0

scala> train.kind
res0: String = ICE

scala> train.number
res1: String = 722
```

Exercise: Add fields to *Train*

- Add (prepend) the class parameter *kind* of type *String* to *Train*
- Make immutable fields out of both class parameters
- Create a *Train* and access the fields

Exercise: Create the class *Time*

- Add the class parameter *hours* of type *Int*
- Add the class parameter *minutes* of type *Int*
- Make immutable fields out of both class parameters
- Add a TODO comment to the class body that we still have to check the preconditions (valid values for hours and minutes)
- Create a *Time* and access the fields

Methods

- Use the keyword *def* to define a method:

```scala
class Time(val hours: Int, val minutes: Int) {
  def minus(that: Time): Int = ...
}
```

- Like fields, methods have public visibility by default:

```scala
scala> val time = new Time(12, 30)
time: Time = Time@32dc51c8

scala> time.minus(new Time(1, 30))
res0: Int = 0
```

Exercise: Implement the method *Time.minus*

- Calculate the difference between the two *Time*s in minutes
- Don't use any helper members yet
- Create a *Time* and verify that the new method is working:

```
scala> val time = new Time(12, 30)
time: Time = Time@113ee167

scala> time.minus(new Time(1, 30))
res0: Int = 660
```

Lazy *vals*

- Use the keyword *lazy* to define an immutable field/variable that is only evaluated on first access:

```scala
lazy val asMinutes: Int = ... // Heavy computation
```

- Why should you use lazy?
  - To reduce initial instantiation time
  - To reduce initial memory footprint
  - To resolve initialization order issues
  - In this course: For didactic reasons ;-)
- But consider the overhead:
  - Guard field
  - Synchronization

Exercise: Improve *Time.minus* using a lazy immutable field

- Add the lazy immutable field *asMinutes* to *Time*
- Use this to simplify the implementation of *minus*
- Verify that *asMinutes* is initialized correctly and *minus* still works:

```
scala> val time = new Time(12, 30)
time: Time = Time@71f08b14

scala> time.asMinutes
res0: Int = 750

scala> time.minus(new Time(1, 30))
res1: Int = 660
```

Operators and operator notation

- Operators are just methods with zero or one parameters:

```
scala> x.+(y) // x == 1 and y == 2
res0: Int = 3

scala> true.unary_!
res1: Boolean = false
```

- You can omit dot and parens, i.e. use operator notation:

```
scala> x + y
res0: Int = 3

scala> !true
res1: Boolean = false
```

Conventions for operator notation

```scala
scala> "Hello " + "World" split " " size
res0: Int = 2
```

- Always use infix notation for symbolic methods (operators)
- Only use infix notation
  - if the method is free of side-effects
  - or if the method takes functions as arguments[7]
- Only use postfix notation
  - if the method is the last operation in a chain of infix calls
  - or for domain specific languages

[7] Coming soon, please have a little patience ;-)

Exercise: Add the operator - to *Time*

- Make it an alias of *minus*, i.e. delegate to *minus*
- Verify that the new operator is working as expected:

```
scala> val time = new Time(12, 30)
time: Time = Time@3513126e

scala> time - new Time(1, 30)
res0: Int = 660
```

Named and default arguments

- You can assign default values to parameters[8]:

```scala
class Time(val hours: Int = 0, val minutes: Int = 0)
```

- Now you can omit trailing arguments:

```scala
scala> val time = new Time(12)
time: Time = Time@2ce628d8
```

- But how can you omit leading arguments? Just use named arguments:

```scala
scala> val time = new Time(minutes = 30)
time: Time = Time@2ce628d8
```

[8] This applies likewise to class and method parameters.

Exercise: Add defaults to *Time*'s parameters

- Add the default value *0* to *hours* and *minutes*
- Try out various combinations of omitting and/or naming arguments for creating a *Time*:

```
scala> val time = new Time()
time: Time = Time@3a3c6542

...
```

Packages

- Use the keyword *package* to declare a package:

```
package org.scalatrain
```

- Looks like Java, but there are differences:
  - Packages truly nest: Members of enclosing packages are visible
  - Package structure and directory structure may differ[9]

[9] Many tools expect equal package and directory structures, though!

Chained package clauses

- A single package clause brings only the last (nested) package into scope:

```scala
package org.scalatrain
class Foo
```

- Use chained package clauses to bring several last (nested) packages into scope; here *Foo* becomes visible without import:

```scala
package org.scalatrain
package util
class Bar extends Foo
```

- Tip: Start with a root package named according to your project and use chained package clauses for your sub-packages

Exercise: Add package clauses

- Add the package clauses to *Train* and *Time*
- Use the “usual” notation
- Use the package name *org.scalatrain*
- Move the files to the directory *src/main/scala/org/scalatrain/*

Imports

- Use the keyword *import* to import a member of a package:

```
import org.scalatrain.Train
```

- Use the underscore to import all members of a package:

```
import org.scalatrain._
```

- Use selector clauses to pick multiple or rename members:

```
import org.scalatrain.{ Time, Train }
import java.sql.{ Date => SqlDate }
```

- You can import members from any "stable identifier", i.e. packages, singleton objects and *val*s

```
val time = new Time(12)
import time._
println(hours)
```

Exercise: Use import clauses

- Try to use *Train* like before introducing packages
- Add a wildcard import clause (use wildcards in the REPL, avoid them in “real” code)
- Try out renaming *Train* with an import selector clause

Access modifiers

- Use the keyword *protected* to make a member only visible inside its enclosing entity as well as its subtypes:

```
class Foo {
  protected val bar = "Bar"
}
```

- Use the keyword *private* to make a member only visible inside its enclosing entity:

```
class Foo {
  private val bar = "Bar"
}
```

Access modifiers

- Use a qualifier to relax access up to the given entity:

```scala
package foo
class Foo {
  private[foo] val bar = "Bar"
}
```

- Use the qualifier *[this]* to restrict access to the instance only:

```scala
class Foo {
  private[this] val bar = "Bar"
}
```

Singleton Objects

- A singleton object is like a class and its sole instance
- Use the keyword *object* to define a singleton object:

```scala
object Foo {
  val bar = "Bar"
}
```

- You can access a singleton object by its name:

```
scala> Foo.bar
res0: java.lang.String = Bar
```

- Singleton objects can be used to replace *static* from Java, but are "real" objects, e.g. can inherit and be passed as arguments

Companion objects

- If a singleton object and a class or trait[10] share the same name, package and file, they are called companions:

```scala
object Time {
  def fromMinutes(minutes: Int): Time = ...
}

class Time(...
```

- From a class or trait you can even access private members of the companion object, e.g. constants or “static” methods
- Except for this, there is no relation at all, especially no “is a”

[10]We will cover traits a little later.

Exercise: Create the companion object for *Time*

- Place it inside the same file like the class *Time*
- Add the method *fromMinutes* taking an *Int* value which creates a *Time* initialized with the given minutes
- "Normalize" the created *Time*:

```
scala> val time = Time.fromMinutes(100)
time: ...Time = org.scalatrain.Time@a51c603

scala> time.hours
res0: Int = 1

scala> time.minutes
res1: Int = 40
```

Meet *Predef*

- The Scala standard library contains the singleton object *Predef*
- Its members are always “silently” imported
- E.g. use the method *require* to check preconditions:

```
scala> require(1 == 2, "This must obviously fail!")
java.lang.IllegalArgumentException: requirement
    failed: This must obviously fail.
```

Exercise: Check hours precondition for *Time*

- Use *require* to check that a *Time* cannot be created with invalid hours, i.e. only with hours greater or equal *0* and less than *24*:

```
scala> new Time(-1, 0)
java.lang.IllegalArgumentException: requirement
    failed: hours must be within 0 to 23!
  ...

scala> new Time(24, 0)
java.lang.IllegalArgumentException: requirement
    failed: hours must be within 0 to 23!
  ...
```

- Keep the TODO comment for the precondition check for minutes!

Case classes

- Use the keyword *case* to define a case class:

```scala
case class Person(name: String)
```

- Now you can create new instances without *new*:

```scala
scala> val person = Person("Joe")
person: Person = Person(Joe)

scala> val person2 = Person.apply("Joe")
person2: Person = Person(Joe)
```

- Look, there is a nice *toString* implementation!

Case class benefits

- And even better, there are nice implementations for *equals* and *hashCode* based on all class parameters:

```
scala> person == person2
res0: Boolean = true
```

- All class parameters are turned into to immutable fields automatically:

```
scala> person.name
res1: String = Joe
```

- There is an easy to use *copy* method using named and default parameters:

```
scala> person.copy(name = "Tim")
res2: Person = Person(Tim)
```

Why are not all classes case classes?

- Sometimes you don't want the overhead
- You cannot (should not) inherit a case class from another one
- Tip: Case classes are perfect "value objects" but in most cases not suitable for "service objects"

Exercise: Case classes

- Turn *Train* and *Time* into case classes
- Remove the *val*s, even if they don't hurt
- Also remove *new* in *Time.fromMinutes*, even if it doesn't hurt
- Try out the varoius case class features:

```
scala> Train("ICE", "722")
res0: org.scalatrain.Train = Train(ICE,722)

scala> Time()
res1: org.scalatrain.Time = Time(0,0)

scala> Time(1, 2)
res2: org.scalatrain.Time = Time(1,2)

scala> Time(1, 2) == Time(1, 3)
res3: Boolean = false
```

Agenda

Testing with specs2

- You could use a Java framework like JUnit or TestNG
- But there are also advanced Scala frameworks
- We choose specs2, because it is very well designed, has deep features and is perfectly maintained

Writing a specification in unit test style

- Extend[11] *org.specs2.mutable.Specification*
- Describe a system under specification followed by *should* and a code block
- Describe an example followed by *in* and a code block returning a *Result*
- Use *todo* to declare a not yet implemented example:

```
class TimeSpec extends Specification {

  "Calling Time.minus" should {
    "return the correct time difference" in {
      todo
    }
    ...
  }
}
```

[11]We will introduce inheritance later!

Writing a specification in acceptance test style

- Extend *org.specs2.Specification* and implement the method *is*
- Chain fragments using the operator ^
- A fragment can be either text or an example
- An example has a description and a *Result*, separated by the operator *!*

```
class TrainSpec extends Specification { def is =

  "Calling Time.minus" ^
    "should return the correct time diff." ! minus ^
    ...
  end

  def minus = todo
  ...
}
```

Running specs2 in Eclipse

- Add a *@RunWith* annotation to the specification
- Use the *org.specs2.runner.JUnitRunner*

```
import org.junit.runner.RunWith
import org.specs2.mutable.Specification
import org.specs2.runner.JUnitRunner

@RunWith(classOf[JUnitRunner])
class TimeSpec extends Specification {
  ...
```

Exercise: Add specs2 as library dependency

- Add the following lines to the build configuration file *build.sbt* in the project directoy *training-scala/*:

```

libraryDependencies ++= Seq(
  "org.specs2" %% "specs2" % "1.8.2" % "test",
  "junit" % "junit" % "4.7" % "test"
)
```

- Attention: Empty lines between settings are important!
- Run the *reload* command in sbt
- Run the *eclipse* command to recreate the Eclipse project files with, then refresh the Eclipse workspace

Exercise: Create the class *TimeSpec*

- For the time being use *todo* to implement the preliminary tests
- Use the directory *src/test/scala/*
- Tests:
  - Preconditions: *hours* and *minutes* must be within 0..23/59
  - Verify the defaults for the class parameters
  - Verify that *Time.minus* works as expected
  - Verify that *Time.asMinutes* is initialized correctly
- Run the tests in sbt with the command *test*
- Run the tests in Eclipse using a JUnit run configuration

Matchers

- Use matchers to define expressive *Results*, e.g.:

```scala
import java.lang.{ IllegalArgumentException => IAE }

Time(-1, 0) must throwAn[IAE]

Time() must equalTo(Time(0, 0))
```

- There are many more, please see the specs2 matchers guide

Exercise: Finalize the tests for *Time*

- Replace the *todo*s with matcher based implementations[12]
- Run the tests and see them fail partially
- Implement the missing precondition (TODO comment) for *Time* until all the tests are passing

[12] Use explicit sample values. The ScalaCheck framework provides test data generation but will not be coverd here.

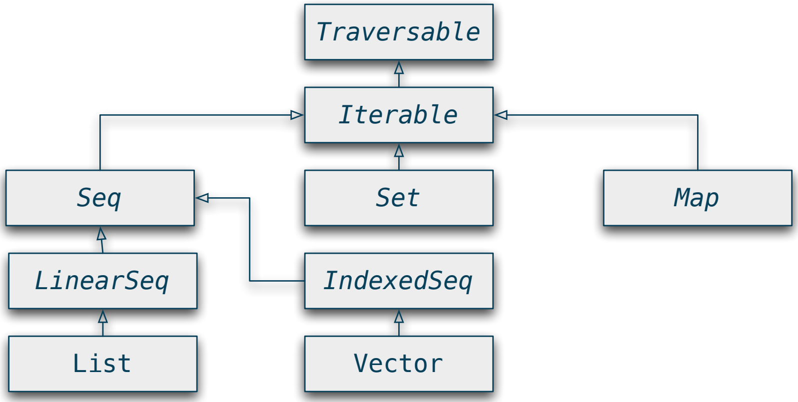
Agenda

- Why Scala?
- Setting up the development environment
- First steps
- Basic OO features
- Testing in Scala
- Collections and functional programming
- For-expressions and for-loops
- Inheritance and traits
- Pattern matching
- XML support

Collections overview

- Scala has a comprehensive collection library
- We will only cover the basics, for details see the very good online documentation

Collection hierachy



Creating collection instances

- Use the “class name” of the collection and append a comma-separated list of items in parens:

```
scala> List(1, 2, 3)
res0: List[Int] = List(1, 2, 3)

scala> Seq(1, 2, 3)
res1: Seq[Int] = List(1, 2, 3)

scala> IndexedSeq(1, 2, 3)
res2: IndexedSeq[Int] = Vector(1, 2, 3)

scala> List(1, 2, "z")
res3: List[Any] = List(1, 2, z)
```

How is that working?

- For each collection type there is a companion object with a factory method *apply*
- Whenever we try to “invoke” an object, the compiler will transform this into calling *apply*:

```
scala> List(1, 2, 3)
res0: List[Int] = List(1, 2, 3)

scala> List.apply(1, 2, 3)
res1: List[Int] = List(1, 2, 3)
```

- Therefore “invoking” an object is just syntactic sugar for ordinary OO method application[13]

[13] This applies in general, not only for collections.

Type parameters

- There are no raw types, all collections are parameterized
- Use square brackets to denote a type parameter[14]:

```scala
case class Parameterized[A](a: A)
```

- Type arguments can be inferred or may be given:

```scala
scala> Parameterized(1)
res0: Parameterized[Int] = Parameterized(1)

scala> Parameterized[Int](1)
res1: Parameterized[Int] = Parameterized(1)
```

[14] This applies not only to classes, but also to traits and methods.

Tuples

- Tuples aren't collections, but important, e.g. to create *Maps*
- Tuples combine a number of objects, each of arbitrary type
- Again, there is syntactic sugar to ease creating Tuples:

```
scala> (1, "a")
res0: (Int, java.lang.String) = (1,a)

scala> Tuple2(1, "a")
res1: (Int, java.lang.String) = (1,a)
```

- Use *_1*, *_2*, etc. to access the first, second, etc. field:

```
scala> res0._1
res2: Int = 1
```

Tuples and *Maps*

- There is an even easier way to create pairs (*Tuple2*s)[15]:

```
scala> 1 -> "a"
res0: (Int, java.lang.String) = (1,a)
```

- Use a list of pairs to create a *Map*:

```
scala> Map(1 -> "a", 2 -> "b")
res0: ...Map[Int,...String] = Map(1 -> a, 2 -> b)
```

[15] This is made possible by implicit conversions which we won't cover in this course.

Immutable and mutable collections

- There are three main packages for collections:
  - *scala.collection*: Abstract base, specialized by the following
  - *scala.collection.immutable*
  - *scala.collection.mutable*
- What does immutability mean?

```
scala> val numbers = Vector(1, 2, 3)
numbers: ...immutable.Vector[Int] = Vector(1, 2, 3)

scala> numbers :+ 4
res0: ...immutable.Vector[Int] = Vector(1, 2, 3, 4)

scala> numbers
res1: ...immutable.Vector[Int] = Vector(1, 2, 3)
```

- Immutable collections aren't mutated in place, but a new instance is returned

Immutable collections by default

- Many collection types can be used without import clauses
- Then the immutable ones are used[16]
- This is enabled by type aliases in the singleton object *Predef* and the package object[17] *scala*

[16] Except for *Seq* where the base one is used.

[17] We won't cover type aliases or package objects in this course.

Some important collection methods

- $++$ appends two collections
- *toSeq*, *toSet*, etc. turns a collection into a specific one
- *isEmpty* and *size* for information regarding size
- *contains* tests whether a collection contains an element
- *head* for the first element, *last* for the last
- *tail* for everything except for the first element, *init* for everything except for the last
- *take* gets the first n elements, *drop* gets all elements except for the first n
- *groupBy* partitions a collection into a *Map* of collections according to some discriminator function

Some more important collection methods

- For *Seq*s: *+:*[18] prepends an element, *:+* appends one
- For *List*s: *::*[19] ("Cons") prepends an element
- For *Map*s: *getOrElse* returns the value for the given key or the given default

[18] Operators ending with *:* are right-associative, i.e. *1 +: Seq(2, 3)* results in *Seq(1, 2, 3)*.

[19] The singleton object *Nil* is the empty *List*, hence *1 :: 2 :: Nil* results in *List(1, 2, 3)*.

Exercise: Add schedule to *Train*

- Create the case class *Station* with the class parameter *name* of type *String* in the file *Train.scala*
- Add the class parameter *schedule* of type *Seq[Station]* to *Train*
- Add a precondition check ensuring that the *schedule* must contain at least two *Station*s (Tip: Use the collection method *size*)
- Create the class *TrainSpec* and add a test for the above precondition

- Collections have a lot of higher order functions[20]
- These take another function as argument or return a function
- Example:

```
scala> val numbers = List(1, 2, 3)
numbers: List[Int] = List(1, 2, 3)

scala> numbers map (x => x + 1)
res0: List[Int] = List(2, 3, 4)
```

- Function literals can also be given in curly braces[21]

[20] Actually collections have methods, but we will use the term higher order functions nevertheless.

[21] We can give any single argument in curly braces.

Function literals

- Just like there are literals for *Int*, *String*, etc. there is also a way to write down anonymous functions
- The compiler will create a function value as an instance of a function type[22]
- Syntax alternatives:

```
scala> numbers map (x => x + 1)
res0: List[Int] = List(2, 3, 4)

scala> numbers map ((x: Int) => x + 1)
res1: List[Int] = List(2, 3, 4)

scala> numbers map (_ + 1)
res2: List[Int] = List(2, 3, 4)
```

[22] We will cover function types shortly.

Function values

- Scala has first-class functions, i.e. functions are objects
- Therefore functions can be assigned to variables and passed as arguments:

```
scala> val addOne = (x: Int) => x + 1
addOne: (Int) => Int = <function1>

scala> numbers map addOne
res3: List[Int] = List(2, 3, 4)
```

Function types

- If functions are objects, which are their types?
- *Int => Int* is syntactic sugar for *Function1[Int, Int]*[23]

```
scala> val addOne: Int => Int = x => x + 1
addOne: (Int) => Int = <function1>
```

- All function types define the method *apply*:

```
scala> addOne(2)
res0: Int = 3

scala> addOne.apply(2)
res1: Int = 3
```

[23] There are also *Function0* to *Function22*.

Turning methods into functions

- Methods aren't objects, they are just members
- Use the underscore to turn a method into a function[24]:

```
scala> def addOne(x: Int) = x + 1
addOne: (x: Int)Int

scala> val f = addOne _
f: (Int) => Int = <function1>
```

- If the signatures of the expected function and the given method match, the compiler can convert it implicitly:

```
scala> numbers map addOne // addOne is a method!
res8: List[Int] = List(2, 3, 4)
```

[24] Actually this is called partial application of functions, but this advanced topic won't be covered.

Important collection methods: *map*

- *map* transforms a collection into a new one[25]:

```scala
trait Traversable[A] {
  def map[B](f: A => B): Traversable[B]
  ...
```

- The type of the collection's elements may change:

```scala
scala> val languages = List("Scala", "JRuby", "Java")
languages: List[...String] = List(Scala, JRuby, Java)

scala> languages map (_.toLowerCase)
res0: List[...String] = List(scala, jruby, java)

scala> languages map (_.length)
res1: List[Int] = List(5, 5, 4)
```

[25] We show simplified versions of the signatures.

Exercise: Add departure times to *Train.schedule*

- Change the type of *Train.schedule* to a sequence of tuples: *Seq[(Time, Station)]*
- Add a TODO comment for the precondition that the schedule must be monotonically increasing in time
- Adjust the test specification (make it compile again)

Exercise: Add the field *stations* to *Train*

- Use *schedule* as a starting point
- Transform its value into a *Seq[Station]* using the collection method *map*
- Add a test verifying that *stations* is initialized correctly

Important collection methods: *flatMap*

- Like *map*, *flatMap* transforms a collection into a new one
- The function argument maps each element to a collection, each of which is expanded into the resulting collection:

```scala
trait Traversable[A] {
  def flatMap[B](f: A => Traversable[B]):
      Traversable[B]
  ...
```

- Comparison to *map*:

```scala
scala> languages map (_.toLowerCase)
res0: List[...String] = List(scala, jruby, java)

scala> languages flatMap (_.toLowerCase)
res1: List[Char] = List(s, c, a, l, a, j, r, u, ...)
```

Exercise: Create the class *JourneyPlanner*

- Add the class parameter *trains* of type *Set[Train]*
- Add the field *stations* to *JourneyPlanner*
- The new field shall contain all *Station*s of all *train*s
- What happens if we use the collection method *map* again?
- Add a test verifying that *stations* is initialized correctly

Important collection methods: *filter*

- *filter* copies selected elements into the resulting collection
- The function argument returns a *Boolean* for each element
- Only elements for which this predicate is *true* are retained:

```scala
trait Traversable[A] {
  def filter(f: A => Boolean): Traversable[A]
  ...
```

Exercise: Add the method *trainsAt* to *JourneyPlanner*

- This new method shall have a parameter of type *Station* and determine all *Train*s that contain that *Station* in their schedule
- Add a test verifying that the correct results are returned for various *Station*s

Agenda

For expressions

- For-expressions are for iteration, but they aren't loops, they yield a collection[26]
- General syntax:

```
for (seq) yield expr
```

- *seq* contains generators, definitions and filters
- *expr* creates an element of the resulting collection

[26]We will see shortly that they work for other data structures, too.

Generators

- Generators drive the iteration
- Common form:

```
x <- coll
```

- *coll* is the collection to be iterated[27]
- *x* is a variable bound to the current element of the iteration[28]
- The (first) generator determines the type of the result:

```
scala> for (i <- List(1, 2, 3)) yield i + 1
res0: List[Int] = List(2, 3, 4)

scala> for (i <- Set(1, 2, 3)) yield i + 1
res1: ...Set[Int] = Set(2, 3, 4)
```

[27] As already mentioned, other data structures can be used, too.

[28] This can be generalized to any pattern, see the upcoming chapter about pattern matching.

Multiple generators

- Either separate multiple generators by semicolon
- Or better use curly braces and new lines:

```
scala> for {
     |   i <- 1 to 3
     |   j <- 1 to i
     | } yield i * j
res0: ...IndexedSeq[Int] = Vector(1, 2, 4, 3, 6, 9)
```

- The inner generators "oscillate" more frequently than the outer ones
- Note: *to* isn't a keyword, but a method[29]

[29] While *to* isn't a member of *Int*, it is made available by implicit conversions, which we won't cover in this course.

Filters

- Filters control the iteration
- Common form:

```
if expr
```

- *expr* must evaluate to a *Boolean*
- Filters can follow generators without semicolon or new line:

```
scala> for {
     |   i <- 1 to 3 if i % 2 == 1
     |   j <- 1 to i
     | } yield i * j
res0: ...IndexedSeq[Int] = Vector(1, 3, 6, 9)
```

- Filter conditions can be written without parens

Definitions

- Definitions are like local *val* definitions
- Common form:

```
x = expr
```

- Definitions can also be directly followed by a filter:

```
scala> for {
     |   time <- times
     |   hours = time.hours if hours > 12
     | } yield (hours - 12) + "pm"
res0: List[String] = List(1pm, 2pm)
```

Exercise: Add the method *stopsAt* to *JourneyPlanner*

- This new method shall have a parameter of type *Station* and determine a *(Time, Train)* for each train that contains the given *Station* in its schedule
- Hint: Use a for-expression with two generators and one filter
- Add a test verifying that the correct results are returned for various *Station*s

Translation of for-expressions

- The compiler translates for-expressions into nested calls of *flatMap*, *map* and *withFilter* (almost like *filter*)

```
for (i <- 1 to 3) yield i + 1
1 to 3 map (i => i + 1)

for (i <- 1 to 3; j <- 1 to i) yield i * j
1 to 3 flatMap (i => 1 to i map (j => i * j))

trains flatMap (train =>
  train.schedule withFilter (timeAndStation =>
    timeAndStation._2 == station
  ) map (timeAndStation =>
    timeAndStation._1 -> train
  )
)
```

For loops

- For-loops return *Unit*, are executed for their side-effects only
- General syntax:

```
for (seq) body
```

- *seq* contains generators, definitions and filters
- *body* may execute a side-effect, its result is omitted

```
scala> for (i <- 1 to 3) println(i)
1
2
3
```

Translation of for-loops

- The compiler translates for-loops into nested calls of *foreach* and *withFilter*

```
for (i <- 1 to 3) println(i)
1 to 3 foreach (i => println(i))
```

Group exercise: Phone mnemonics

- Task: Given a mapping from phone keys to mnemonics and a dictionary, write a program that translates a phone number into all possible phrases made up from words in the dictionary
- Example: 7225276257 should be translated to “Scala rocks”
- Taken from Lutz Prechelt, “An Empirical Comparison of Seven Programming Languages”[30]
- Tested with Tcl, Python, Perl, Rexx, Java, C, C++
- About 100 LOC for scripting languages, 200-300 for others
- Let’s see whether we can do better with Scala!

[30] IEEE Computer 33 (10): 23-29 (2000)

Group exercise: Phone mnemonics - outline

```scala
class PhoneMnemonics(words: Set[String]) {

  val mnemonics = Map('2' -> "ABC", '3' -> "DEF", ...)

  val charCode: Map[Char, Char] = Map.empty

  def wordCode(word: String): String = ""

  val wordsForNumber: Map[String, Set[String]] = Map.empty

  def encode(number: String): Set[Seq[String]] = Set.empty

  def translate(number: String): Set[String] =
    encode(number) map { _ mkString " " }
}
```

Agenda

Class inheritance

- Use the keyword *extends* to define a subclass of another class:

```scala
class Animal
class Bird extends Animal
```

- Omitting *extends* means *extends AnyRef*[31]

[31] *AnyRef* is equivalent to *java.lang.Object* on the JVM.

Calling the superclass constructor

- Subclasses must immediately call their superclass constructor:

```
class Animal(val name: String)
class Bird(name: String) extends Animal(name)
```

- This won't compile:

```
scala> class Animal(val name: String)
defined class Animal

scala> class Bird(name: String) extends Animal
<console>:8: error: not enough arguments ...
Unspecified value parameter name.
```

Final classes

- Use the keyword *final* to prevent a class from being subclassed:

```
scala> final class Animal
defined class Animal

scala> class Bird extends Animal
<console>:8: error: illegal inheritance from final
    class Animal
```

Sealed classes

- Use the keyword *sealed* to allow subclassing only within the same source file:

```
sealed class Animal
class Bird extends Animal
class Fish extends Animal
```

- This means, that sealed classes can only be subclassed by you, but not by others, i.e. you know all subclasses[32]

[32] As we will see soon, this is a valuable information for pattern matching.

Example: Enumerations

- Create a singleton object extending from *Enumeration*:

```
object Consumers extends Enumeration {
  val Herbivore = Value
  val Carnivore = Value
}
```

- Each element of the enumeration is defined as an immutable field initialized by calling *Value*[33] with an optional name
- The enumeration's elements have an *id* and a name:

```
scala> Consumers.values map { value =>
     |   value.id -> value.toString
     | }
res0: ... = Set((0,Herbivore), (1,Carnivore))
```

[33] The values' type is path dependent: *Consumer.Value*

Exercise: Use an enumeration for *Train.kind*

- Create the enumeration *TrainKind* with three elements[34]:
  - *Ice*[35] with the name *"ICE"*
  - *Re* with the name *"RE"*
  - *Brb* with the name *"BRB"*
- Change the type of *Train.kind* from *String* to *TrainKind.Value*
- Adjust the test cases, i.e. make the whole project compile again

[34] "ICE", "RE" and "BRB" are (some) kinds of trains in Germany.
[35] Constants are written in upper camel case (first letter capitalized).

Overriding members

- Use the keyword *override* to override a superclass member:

```
class Animal {
  val name = "Animal"
}
class Bird extends Animal {
  override val name = "Bird"
}
```

- *override* is mandatory to avoid mistakes:

```
class Bird extends Animal {
  override val nam = "Bird"
}

<console>:9: error: value nam overrides nothing
```

- Use the keyword *final* to prevent a member from being overridden

Overriding methods with immutable variables

- You can override a parameterless method with an immutable variable:

```scala
class Animal {
  def name = "Animal"
}
class Bird extends Animal {
  override val name = "Bird"
}
```

- The other way round is not possible!

Exercise: Override *Time.toString*

- As *Time* is a case class, the result of *toString* looks already quite nice
- But we can do better: Let's use string formatting with the format *"%02d:%02d"* to get something like *"12:55"*
- Hint: Scala adds the instance method *format* to *String*; simply apply it to the above format string: *"%02d:%02d".format(...)*
- First use a method and then switch to an immutable field
- Add a test verifying that *Time.toString* returns a correctly formatted result

Abstract classes

- Use the keyword *abstract* to define an abstract class
- Simply omit the initialization or implementation to define an abstract field or method:

```scala
abstract class Animal {
  val name: String
  def hello: String
}
```

- *abstract* is mandatory to prevent you from making a class abstract by mistake

Implementing abstract members

- Initialize or implement an abstract field or method to make it concrete:

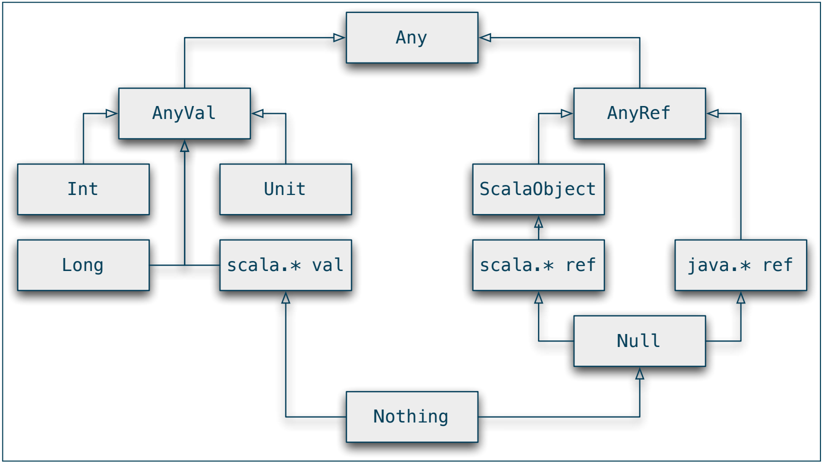
```scala
class Bird(override val name: String) extends Animal {
  override def hello = "Beep"
}
```

- While using *override* to initialize/implement an abstract member isn't mandatory, it is recommended to prevent you from mistakes
- There is one exception: Don't use *override val* for case class parameters, because case classes should never be subclassed

Exercise: Refactor *Train* to use a sealed abstract class

- Create the sealed abstract class *TrainInfo* with the abstract def *number* of type *String*
- Add the case classes *Ice*, *Re* and *Brb* extending *TrainInfo*
- Add the class parameter *hasWifi* of type *Boolean* with default value *false* to *Ice*
- Replace *Train*'s fields *kind* and *number* with the single field *info* of type *TrainInfo*
- Delete the enumeration *TrainKind*
- Adjust the test cases, i.e. make the whole project compile again

Scala type hierarchy



Group exercise: Two-dimensional layout library

- We want to build and render two-dimensional layout elements where element represents a rectangle filled with text
- We want to create elements using factory methods
- We want to combine elements horizontally and vertically:

```
scala> val e1 = Element("Hello") above Element("***")
...
scala> val e2 = Element("+++") above Element("World")
...
scala> e1 beside e2
res0: layout.Element =
Hello +++
 *** World
```

Traits

- Use the keyword *trait* to define a trait:

```scala
trait Swimmer {
  def swim = "I am swimming!"
}
```

- Traits encapsulate fields and methods[36]
- Traits are abstract and have no parameters[37]
- Traits can explicitly inherit from a class:

```scala
class A
trait B extends A
```

[36] You can look at traits as interfaces with concrete members.
[37] You can also look at traits as abstract classes without parameters.

The use case for traits

- As in Java, there is no multiple (class) inheritance in Scala:

```scala
abstract class Animal {
  val name: String
}
class Bird(val name: String) extends Animal {
  def fly = "I am flying!"
}
class Fish(val name: String) extends Animal {
  def swim = "I am swimming!"
}
class Duck // ??
```

- How can we avoid code duplication?

Mix-in composition

- Use the keyword *with* to mix a trait into a class that already extends another class:

```
class Fish(val name: String) extends Animal with
    Swimmer
class Duck(name: String) extends Bird(name) with
    Swimmer
```

- Use the keyword *extends* to mix a trait into a class that doesn't explicitly inherit from another class:

```
trait A
class B extends A // class B extends AnyRef with A
```

- Mix-ins are like multiple inheritance just without the issues[38]

[38] The Scala compiler is able to linearize all inherited classes and mixed-in traits.

Mixing-in multiple traits

- Use the keyword *with* repeatedly to mix-in multiple traits:

```
trait A
trait B
trait C
class D extends A with B with C
```

- If multiple traits define the same members, the outermost (rightmost) one "wins"

Mix-in rules

- Traits must respect the inheritance hierarchy:

```
scala> class A; class B; trait C extends A
...

scala> class D extends B with C
<console>:10: error: illegal inheritance; superclass B
 is not a subclass of the superclass A of the mixin
     trait C
```

- Concrete members must be overridden:

```
scala> trait A { def x = 1 }; trait B { def x = 2 }
...

scala> class C extends A with B
<console>:9: error: overriding method x in trait A ...
 ... needs `override' modifier
```

Exercise: Mix *Ordered* into *Time*

- This let's you compare *Time*s using $>$, $>=$, etc.
- Implement the abstract method *compare*
- Add a test verifying that *Time*s are ordered correctly

Agenda

- Why Scala?
- Setting up the development environment
- First steps
- Basic OO features
- Testing in Scala
- Collections and functional programming
- For-expressions and for-loops
- Inheritance and traits
- Pattern matching
- XML support

Match expressions

- Match expressions look a little like Java's *switch*, but they are different and much more powerful
- General syntax:

```
expr match {
  case pattern1 => result1
  case pattern2 => result2
  ...
}
```

- *expr* in front of the keyword *match* is an arbitrary expression
- The result of *expr* is matched against the various alternatives inside the body of *match*
- Matching happens from top to bottom

Match alternatives

- General syntax:

```
case pattern => result
```

- An alternative starts with the keyword *case*
- *pattern* is one of various pattern types[39]
- *result* is an arbitrary expression[40]
- If a pattern matches, the match expression returns *result*
- If no alternative matches, a *MatchError* is thrown

[39] We will discuss the various pattern types shortly.

[40] The expression can span multiple lines without using curly braces.

Wildcard pattern

- Use the underscore as a wildcard to match everything:

```scala
case _ => result
```

- Use the wildcard pattern as the last alternative to prevent *MatchError*s:

```scala
def whatIsIt(any: Any) = any match {
  case _ => "Something unknown"
}

scala> whatIsIt(Time())
res0: java.lang.String = Something unknown
```

Constant pattern

- Use a "stable identifier" to match "something constant":

```
def whatIsIt(any: Any) = any match {
  case "12:00" => "High noon"
  case _ => "Something unknown"
}

scala> whatIsIt("12:00")
res0: java.lang.String = High noon

scala> whatIsIt("12:01")
res1: java.lang.String = Something unknown
```

- Stable identifiers are literals and *val*s or singleton objects starting with a capital letter
- Alternatively enclose such an identifier in backticks when starting with a small letter

Variable pattern

- Use a variable starting with a small letter to capture a value:

```
def whatIsIt(any: Any) = any match {
  case x => "Something: " + x
}

scala> whatIsIt(Time())
res0: java.lang.String = Something: 00:00
```

- The variable pattern used by itself will match everything

Typed pattern

- Use a type annotation to match certain types only:

```scala
def whatIsIt(any: Any) = any match {
  case x: String => "A String: " + x
  case _: Int => "An Int value"
  case _ => "Something unknown"
}

scala> whatIsIt("12:01")
res01: java.lang.String = A String: 12:01

scala> whatIsIt(1)
res1: java.lang.String = An Int value
```

- The typed pattern is always combined with the wildcard or variable pattern

Tuple pattern

- Use tuple syntax to match and decompose tuples:

```
def whatIsIt(any: Any) = any match {
  case ("12:00", "12:01") => "12:00..12:01"
  case ("12:00", x) => "High noon and " + x
  case _ => "Something else"
}

scala> whatIsIt("12:00" -> "midnight")
res0: java.lang.String = High noon and midnight
```

- The tuple pattern is combined with other patterns, e.g. with the constant or variable pattern

Constructor pattern

- Use constructor syntax to match and decompose case classes:

```scala
def whatIsIt(any: Any) = any match {
  case Time(12, 00) => "High noon"
  case Time(12, minutes) => "12:%02d" format minutes
}

scala> whatIsIt(Time(12, 01))
res0: java.lang.String = 12:01
```

- The constructor pattern is combined with other pattern, e.g. with the constant or variable pattern or with deeply nested constructor patterns

Sequence pattern

- Use “sequence constructors” to match and decompose sequences:

```scala
def whatIsIt(any: Any) = any match {
  case Seq(1, 2) => "1, 2"
  case Seq(1, 2, _*) => "1, 2, ..."
  case Seq(Time(h, m), x) =>
    "%02d:%02d, %s".format(h, m, x)
}
```

- _* wildcard matches a trailing subsequence
- The sequence pattern is also combined with other patterns

Pattern guards

- Combining patterns gives you a lot of control over matching, but sometimes that's just not enough
- Use the keyword *if* to define a pattern guard:

```
def whatIsIt(any: Any) = any match {
  case s: String if s startsWith "x" => "x..."
  case _: String => """Not starting with "x"!"""
  case _ => "Something else"
}

scala> whatIsIt("xyz")
res0: java.lang.String = x...
```

- Pattern guards can be written without parens

Exercise: Add the method *isShortTrip* to *JourneyPlanner*

- A trip between two *Station*s is a short trip, if:
  - There exists a connection with a single train
  - There is at most one *Station* between the given two
- *isShortTrip* has the parameters *from* and *to* of type *Station*
- Impementation hint: Take a look at the collection methods *exists* and *dropWhile* and use pattern matching with the sequence pattern
- Add a test case verifying that short trips are calculated correctly

Catching exceptions

- There are no checked exceptions in Scala
- Use patterns to catch exceptions:

```
try {
  // Possibly throwing a NumberFormatException
} catch {
  case e: NumberFormatException => ...
}
```

Patterns outside of match expressions

- Use patterns in *val* definitions or generators:

```
scala> val (morning, highNoon) = Time(6) -> Time(12)
morning: org.scalatrain.Time = 06:00
highNoon: org.scalatrain.Time = 12:00

scala> val charAndIndexList = List('a' -> 1, 'b' -> 2)
charAndIndexList: ... = List((a,1), (b,2))

scala> for ((char, index) <- charAndIndexList) {
     |   println("%s: %s".format(index, char))
     | }
1: a
2: b
```

Exercise: Use patterns to improve readability

- Replace clumsy tuple element accessors by patterns:
- Refactor *JourneyPlanner.stopsAt*
- Refactor *PhoneMnemonics.charCode*

Agenda

XML literals

- XML literals[41] are baked into the language:

```
scala> <time hours="12" minutes="00"/>
res0: ...Elem = <time minutes="00" hours="12"></time>
```

- The compiler checks whether XML literals are well-formed:

```
scala> <time><unclosed></time>
<console>:1: error: in XML literal: expected closing tag of unclosed
```

[41] We will cover important types of the XML library shortly.

Insert Scala code into XML literals

- Use curly braces to insert Scala expressions into XML literals:

```
scala> <random>{ Random.nextInt }</random>
res0: scala.xml.Elem = <random>-1249076074</random>
```

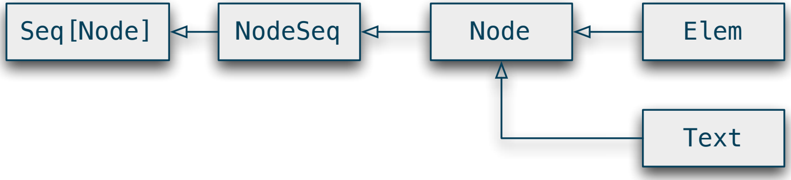
- For text nodes any type works, but for attribute nodes you have to provide *String*s[42]:

```
scala> val time = Time(12)
time: org.scalatrain.Time = 12:00

scala> <time hours={ time.hours.toString }/>
res0: scala.xml.Elem = <time hours="12"></time>
```

[42] Actually you have to provide a *NodeSeq* in both cases, but only for text nodes any type is converted into a *String* first.

Important types of the XML library



- *NodeSeq* is the root of all XML types
- *Elem* represents XML elements
- *Text* represents text nodes

XML objects are sequences

- Use ++ to add two *NodeSeq*s:

```
scala> val ab = <a/> ++ <b/>
ab: scala.xml.NodeSeq = NodeSeq(<a></a>, <b></b>)
```

- Use *child* to access the child nodes of an *Elem*:

```
scala> val xml = <a><b><c/></b><b><c/></b></a>
xml: ...Elem = <a><b><c></c></b><b><c></c></b></a>

scala> xml.child
res0: .. = ArrayBuffer(<b><c></c></b>, <b><c></c></b>)

scala> xml.child.head
res1: scala.xml.Node = <b><c></c></b>
```

XPath like queries

- Use the operator \ to query an *Elem* for children:

```
scala> xml \ "b"
res0: scala.xml.NodeSeq = NodeSeq(<b><c></c></b>,
    <b><c></c></b>)
```

- Use the operator \\ to query an *Elem* for descendants:

```
scala> xml \\ "c"
res0: scala.xml.NodeSeq = NodeSeq(<c></c>, <c></c>)
```

- Use @ to query for attributes:

```
scala> val xml = <a><b name="b1"/><c name="b2"/></a>
xml: .. = <a><b name="b1"></b><c name="b2"></c></a>

scala> xml \\ "@name"
res0: scala.xml.NodeSeq = NodeSeq(b1, b2)
```

Digression: The *Option* class

- In Java an optional result is typically expressed by *null*
- In Scala we have the abstract class *Option* that can either be *Some* wrapping a value or *None*:

```
scala> Option(1)
res0: Option[Int] = Some(1)

scala> Option(null)
res1: Option[Null] = None
```

- Benefits: No more forgetting *if (... == null)*
- *Option* defines *map*, *flatMap*, *withFilter*, etc., i.e. can be used in for-expressions

Exercise: Add XML serialization to *Time*

- Add the method *toXml* to *Time*:

```
<time hours="12" minutes="01" />
```

- Add the method *fromXml* to the companion object *Time*:
  - Add one parameter of type *Elem*
  - The result type shall be an *Option[Time]*, depending whether the given *Elem* can be "parsed"
  - Implementation hint: Use *Exception.catching* from the package *scala.util.control* to treat *NumberFormatException*s
- Add tests, including one "round trip" serialization/deserialization